

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Інститут прикладного системного аналізу

Кафедра системного проектування

Курсова робота

з дисципліни «Алгоритмізація та програмування»

на тему: «Застосування бінарних дерев»

Виконав:

студент I курсу, групи ДА-01

Дячина Микита Олегович

Керівник:

доцент, к.т.н.

Безносик Олександр Юрійович

Засвідчую, що у цій дипломній роботі немає
запозичень з праць інших авторів без
відповідних посилань.

Студент _____

Київ – 2021 рік

ІПСА КПІ ім. І. Сікорського
(назва вищого закладу освіти)

Кафедра	Системного проектування
---------	-------------------------

Дисципліна	Алгоритмізація та програмування
------------	---------------------------------

Спеціальність	122 Комп'ютерні науки
---------------	-----------------------

Курс	1	Група	ДА-01	Семестр	2
------	---	-------	-------	---------	---

ЗАВДАННЯ на курсовий проект(роботу) студента

Дячини Микити Олеговича
(прізвище, ім'я, по батькові)

1. Тема проекту(роботи)	Застосування бінарних дерев.

2. Строк здачі студентом закінченого проекту(роботи)	25.05.2021
------------------------------------------------------	------------

3. Вихідні дані до проекту(роботи)	Мова програмування C++, пакет інструментів CMake, редактор коду Visual Studio Code, алгоритми сортування бінарним деревом та купою, алгоритм Хафмана.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці)	
<i>Постановка задачі.</i>	
<i>Метод розв'язку задачі</i>	
<i>Опис програмного продукту.</i>	
<i>Висновки.</i>	
<i>Список літератури.</i>	
<i>Додаток А. Текст програми.</i>	

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)	
<i>Загальна блок-схема алгоритму.</i>	
<i>Ілюстрації роботи програми.</i>	

6. Дата видачі завдання	19.03.2021
-------------------------	------------

КАЛЕНДАРНИЙ ПЛАН

№/п	Назва етапів курсового проекту (роботи)	Строк виконання етапів роботи	Примітка
1.	Вибір теми курсової роботи. Опрацювання відповідної літератури. Оформлення листа Завдання.	2, 3-й тиждень лютого	
2.	Аналіз постановки задачі.	3, 4-й тиждень лютого	
3.	Вибір та дослідження методів, вибір відповідних структур даних, розробка алгоритму. Перше узгодження з керівником.	1-й тиждень березня	
4.	Проектування інтерфейсу.	2-й тиждень березня	
5.	Друге узгодження з керівником.	2-й тиждень березня	
6.	Програмна реалізація.	3, 4-й тижні березня	
7.	Демонстрація першого варіанту. Трете узгодження з керівником.	1-й тиждень квітня	
8.	Заключне тестування програми.	2, 3-й тижні квітня	
9.	Аналіз результатів. Оформлення звіту.	до 2-го тижня травня	
10.	Захист та демонстрація курсової роботи.	до 25.05 – «А» до 30.05 – «В,С» до 05.06 – «D,E»	

Студент		Дячина Микита Олегович
	(підпис)	

Керівник			Безносик Олександр Юрійович
	(підпис)		(прізвище, ім'я, по батькові)

25.05.2021		
(дата)		

ЗМІСТ

ВСТУП.....	5
РОЗДІЛ 1. ІНСТРУМЕНТИ РОЗРОБКИ. ПОСТАНОВКА ЗАДАЧ.....	6
1.1. Обґрунтування вибору інструментів розробки.....	6
1.2. Бінарні дерева.....	7
1.3. Постановка задач проектування.....	7
РОЗДІЛ 2. АНАЛІЗ ТА РЕАЛІЗАЦІЯ АЛГОРИТМІВ.....	9
2.1. Визначення способів реалізації алгоритмів.....	9
2.2. Меню.....	10
2.3. Heapsort.....	12
2.4. Сортування двійковим деревом.....	14
2.5. Пошук.....	17
2.6. Алгоритм Хафмана.....	19
РОЗДІЛ 3. ОПИС ПРОГРАМНОГО ПРОДУКТУ.....	22
3.1. Опис меню програми.....	22
3.2. Можливі помилки.....	23
3.3. Структура коду програми, спосіб компіляції та запуску.....	24
ВИСНОВКИ.....	26
СПИСОК ЛІТЕРАТУРИ.....	27
ДОДАТКИ.....	28
Додаток А. Текст програми.....	28

ВСТУП

Бінарні дерева є одними з найважливіших нелінійних структур даних. Їх дуже часто використовують у програмуванні як представлення для множин, як основу для баз даних, у багатьох алгоритмах. Деякі з цих алгоритмів реалізовані в цій роботі. Майже у всіх сучасних мовах програмування в стандартних бібліотеках є бінарні дерева у тому чи іншому вигляді.

Виконання даної роботи розвиває навички програмування мовою C++ та реалізації вивчених протягом курсу структур даних та алгоритмів.

Метою даної роботи є розробка найвідоміших алгоритмів з використанням бінарних дерев та закріплення набутих знань та практичних навичок.

Завдання роботи: на основі аналізу літературних джерел дослідити особливості бінарних дерев та алгоритми на їх основі, довести доцільність їх використання порівняно з іншими структурами даних, описати їх переваги та недоліки, навчитися застосовувати їх у програмному забезпеченні для виконання прикладних задач, створити зручний інтерфейс використовуючи засоби C++ на основі емулятора терміналу.

При виконанні роботи для реалізації програмного забезпечення було використано кросплатформне середовище розробки з відкритим кодом Visual Studio Code, компілятор C-подібних мов Clang та пакет утиліт для автоматичної побудови програм CMake, що функціонували на платформі Ubuntu 20.04, для пошуку інформації в мережі Інтернет — веб-браузер Mozilla Firefox, для оформлення тексту пояснювальної записки — Libreoffice Writer.

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

Інформаційною основою дослідження є чинні міжнародні стандарти та Державні стандарти України в галузі інформаційних технологій та наукових досліджень, праці науковців та інженерів-практиків, Інтернет-джерела та інша, в тому числі, навчальна література.

РОЗДІЛ 1. ІНСТРУМЕНТИ РОЗРОБКИ. ПОСТАНОВКА ЗАДАЧ

1.1. Обґрунтування вибору інструментів розробки

Для написання коду розглядалися популярні середовища розробки, такі як:

- Visual Studio Code – розширюваний кросплатформний редактор коду, який добре інтегрується з компіляторами, інтерпретаторами та інструментами для налагодження багатьох мов програмування, у тому числі C++. Працює досить швидко, оскільки встановлюються тільки ті розширення, які потрібні програмісту.
- Microsoft Visual Studio – повнофункціональне середовище розробки тільки для Windows. Підтримує багато мов програмування, таких як: C, C++, Python, C#, F#, Visual Basic, JavaScript, ... Є можливість комбінувати різні пакети інструментів для розробки програмного забезпечення, підтримується інтеграція з Unity, Android SDK, ... Через свою потужність використовується дуже широко, але й набагато довше йде процес встановлення, потребує набагато більше пам'яті та ресурсів комп'ютера для роботи.
- Code::Blocks – кросплатформне середовище розробки з відкритим кодом для мов C/C++ і Fortran. Як і Microsoft Visual Studio, є повнофункціональним середовищем розробки, хоча кількість функціоналу та підтримуваних інструментів набагато менша.
- CLion – повнофункціональне кросплатформне середовище розробки з підтримкою C/C++, Rust, Swift, Python, Fortran, Javascript, ... та з розумною системою редагування коду. Не має безкоштовної версії.

З урахуванням ціни, зручності редагування та налагодження коду, кросплатформності та швидкості роботи, було обрано Visual Studio Code з пакетом інструментів для компіляції C/C++ CMake, компілятором Clang і розширенням для автодоповнення коду на C/C++.

1.2. Бінарні дерева

У комп'ютерних науках бінарне дерево — це така структура даних, яка складається з вузлів з ключами та даними. Вузли порівнюються за ключами та мають до двох дочірніх вузлів (нащадків). Один з дочірніх вузлів вважають лівим, а інший — правим. Один з вузлів дерева є кореневим і не має батьківського вузла, а всі решта вузлів є нащадками кореневого. Вузли без нащадків називають листами [1].

У даній роботі для різних алгоритмів будуть використовуватися різні види бінарних дерев. Бінарна купа є частково впорядкованою структурою даних і найбільш ефективно працює тоді, коли потрібно часто витягати найбільший елемент з дерева. Бінарне дерево пошуку є впорядкованою структурою і корисна для швидкого пошуку елементів. Збалансовані бінарні дерева забезпечують оптимальну асимптотичну складність для бінарних дерев пошуку.

Бінарні дерева - це структури даних з дуже широкою сферою використання, тому в даній роботі було обрано 4 найбільш поширених алгоритми на основі бінарних дерев для дослідження та реалізації.

1.3. Постановка задач проектування

Виходячи з теми курсової роботи та найбільш вживаних алгоритмів з використанням бінарних дерев, для даної роботи я визначив наступні задачі:

- дослідити застосування бінарних дерев;
- реалізувати алгоритми:
 - алгоритм Хафмана;
 - сортування бінарним деревом;
 - пошук у бінарних деревах;
 - сортування купою;
- реалізувати лаконічний консольний інтерфейс;
- перевірити програму на помилки за допомогою контрольних прикладів;

- проаналізувати складність реалізованих алгоритмів;
- дослідити переваги та недоліки бінарних дерев порівняно з іншими структурами даних;
- порівняти різні типи збалансованих дерев;
- створити опис основної програми, окремих програмних модулів та інтерфейсу;
- визначити можливості для вдосконалення роботи.

РОЗДІЛ 2. АНАЛІЗ ТА РЕАЛІЗАЦІЯ АЛГОРИТМІВ

2.1. Визначення способів реалізації алгоритмів

Для початку визначимо, які структури даних буде використано для різних алгоритмів. Це може бути одне зі збалансованих бінарних дерев пошуку, незбалансоване бінарне дерево пошуку, бінарна купа тощо.

В алгоритмові Хафмана префіксне дерево не буде збалансованим, тому тут неможливо використовувати одне зі збалансованих бінарних дерев.

Для алгоритму tree sort також використано незбалансоване бінарне дерево. Середня складність цього алгоритму — $O(n \log n)$, у найгіршому випадку — $O(n^2)$ (якщо дерево вироджене). Для гарантії складності $O(n \log n)$ у найгіршому випадку можна реалізувати цей алгоритм на основі збалансованого бінарного дерева.

Пошук у бінарних деревах дуже часто використовується у різних проектах, оскільки він відбувається за логарифмічний час $O(\log n)$ і допускає швидкий пошук одразу проміжку елементів, на відміну від хеш-таблиць, які забезпечують швидке отримання одного елемента, але не передбачають пошук усіх елементів таблиці на деякому проміжку.

Для алгоритму сортування heapsort використана бінарна купа на основі масиву. Це можливо за рахунок того, що бінарна купа завжди є повним бінарним деревом. Така реалізація не потребує додаткової пам'яті для зберігання вказівників, оскільки всі елементи можна отримати за допомогою арифметичних операцій над індексами.

Щодо інтерфейсу, то це буде консоль, у яку виводиться меню з нумерацією можливих варіантів вибору. Якщо введене число не відповідає жодному з існуючих варіантів вибору, або введений символ не є числом, то виводиться повідомлення про помилку та пропонується зробити вибір ще раз. Після вибору алгоритму для тестування, у користувача запрошується шлях до файлу (для алгоритму Хафмана), або масив чисел (для алгоритмів сортування та пошуку).

Кожен крок супроводжується поясненнями, інтерфейс повинен бути зрозумілим і не заплутувати користувача.

2.2. Меню

Основне меню програми реалізовано доволі стандартним методом. Є змінна символьного типу, в яку вводиться символ з клавіатури у нескінченному циклі. Перед зчитуванням кожного символу виводиться пронумерований перелік варіантів вибору. Після кожного зчитування виконується switch для введеного символу, який викликає відповідну функцію залежно від введеного числа, або виводить повідомлення про помилку. Блок-схема роботи меню наведена на рисунку 1.

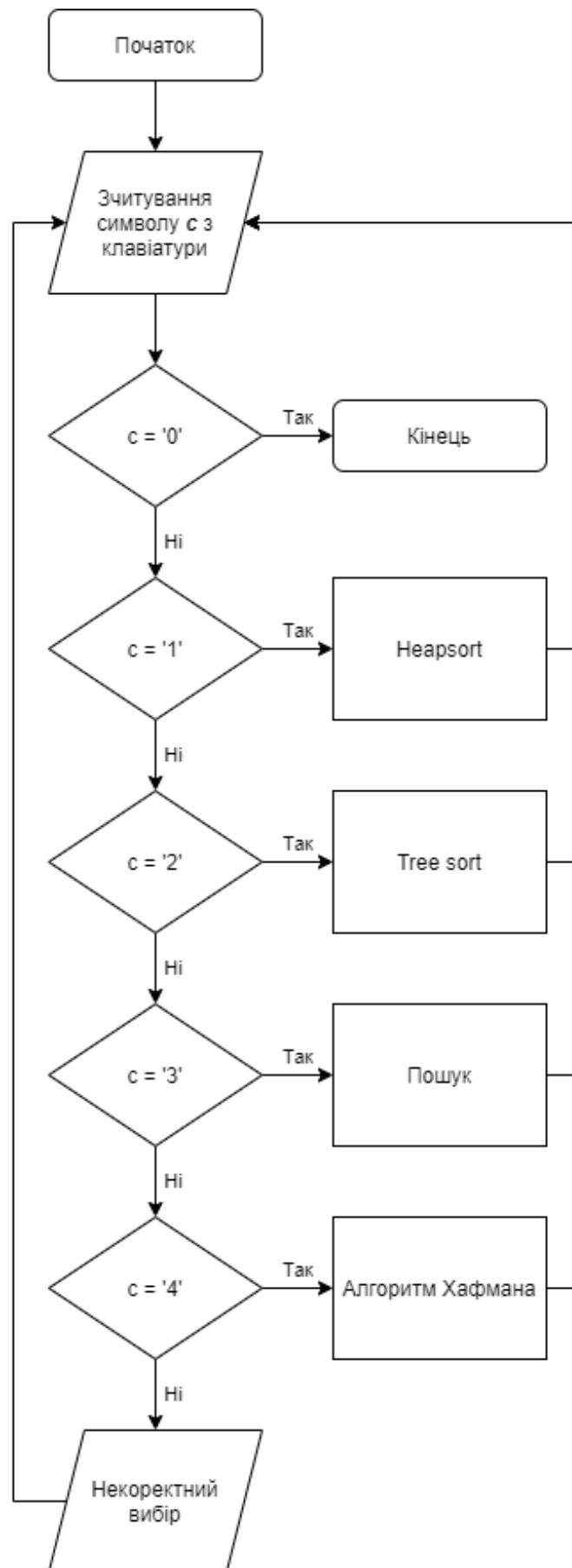


Рис. 1 — Блок-схема алгоритму меню програми

2.3. Heapsort

Бінарна купа (англ. Binary heap) – це така структура даних, яку можна представити у вигляді бінарного дерева, при чому для всіх вузлів A ключ їхнього батьківського вузла $P < A$ (для min-heap), або $P > A$ (для max-heap). Таку структуру даних можна вважати частково впорядкованою, і вона ефективна для задач, у яких потрібно повторно доставати найбільший (або найменший) елемент багато разів. Найчастіше ці структури даних реалізують на масивах, оскільки одною властивістю бінарної купи є повнота — кожен рівень бінарного дерева повинен заповнюватися зліва направо, і якщо існує вузол A на рівні X , то A повинен бути кореневим вузлом або рівень $X-1$ має бути повністю заповненим.

Наприклад, якщо на четвертому рівні буде 1 вузол, то на першому повинен бути 1 вузол (кореневий), на другому рівні повинно бути 2 вузла (два дочірніх вузла кореня), на третьому рівні — 4 вузла (по два дочірніх для двох вузлів другого рівня). Таким чином, можна однозначно проіндексувати всю бінарну купу і простими арифметичними операціями над індексами отримувати індекси батьківських та дочірніх вузлів. Ось як це реалізовано у коді (з урахуванням того, що у мові C++ індексація масиву починається з 0):

```
int parent(int i)
{
    return (i - 1) / 2;
}

int left(int i)
{
    return i * 2 + 1;
}

int right(int i)
{
    return i * 2 + 2;
}
```

Одне з застосувань цієї структури даних — сортування heapsort. Воно базується на тому, що будь-який масив можна сприймати за бінарну купу і використати функцію max-heapify, яка підтримує основну властивість бінарної

купи, для всіх елементів від останнього елемента передостаннього рівня до кореня [2].

Наприклад, якщо масив складається з 6 чисел, то 1 з них — корінь бінарної купи, потім 2 числа на другому рівні і 3 числа на третьому рівні. Тому ми починаємо з останнього числа на другому рівні і перевіряємо, чи підтримується основна властивість бінарної купи для цього елемента і його дочірніх елементів. Якщо дочірні елементи більші за даний, то потрібно поміняти місцями більший з дочірніх елементів і даний елемент. Далі потрібно рекурсивно викликати `max-heapify` для позиції дочірнього елемента, який тепер став на місце аргументу функції `max-heapify`. Коли ми викличемо цю функцію для всіх елементів масиву від останнього елемента передостаннього рівня до кореня, то даний масив стане бінарною купою.

Після описаної процедури (`build-max-heap`) ми можемо послідовно витягувати найбільший елемент з купи і записати їх у масив, отримавши його у відсортованому вигляді. Але це сортування можна виконати навіть краще — `in-place`, тобто масив можна відсортувати не виділяючи додаткової пам'яті. Для цього потрібно послідовно міняти місцями перший елемент купи (який є найбільший) з останнім, а потім переставати вважати останній елемент масиву за частину купи (тобто зменшити її розмір). Після кожної ітерації потрібно викликати `max-heapify` на кореневому вузлі. Таким чином ми отримаємо відсортований масив, оскільки ми послідовно переміщували найбільший елемент на останню позицію масиву, потім другий найбільший на передостанню позицію і т.д.

Асимптотична складність функції `max-heapify` — $O(\log n)$, оскільки в найгіршому випадку нам потрібно пройти по всій висоті дерева. Асимптотична складність `build-max-heap` — $O(n \log n)$, оскільки ми викликаємо `max-heapify` n разів. Складність `heapsort` — $O(n \log n) + O(n) = O(n \log n)$, оскільки спочатку ми викликаємо `build-max-heap`, а потім лінійно проходимо циклом по всьому масиву.

Описаний алгоритм сортування не потребує багато додаткової пам'яті та завжди має оптимальну асимптотичну складність $O(n \log n)$, що робить його популярним рішенням для систем з обмеженою оперативною пам'яттю. Широко використовується в ядрі Linux.

2.4. Сортування двійковим деревом

Ще одною широко використаною структурою даних є бінарне дерево пошуку. Це таке бінарне дерево, яке забезпечує пошук елемента або всіх елементів інтервалу за $O(\log n)$. Його основна властивість у тому, що для будь-якого елемента R та його лівого й правого дочірніх елементів X та Y , $X < R < Y$. При чому немає обмежень на повноту, як для бінарної купи [3].

Але крім пошуку це також дозволяє сортувати масиви даних, просто послідовно додавши їх до цього дерева (алгоритм додавання елемента наведено на рис. 2 у вигляді блок-схеми), оскільки при рекурсивному проходженні цього дерева по порядку всі елементи будуть відсортовані [4]. Блок-схема цього алгоритму наведена на рис. 3.

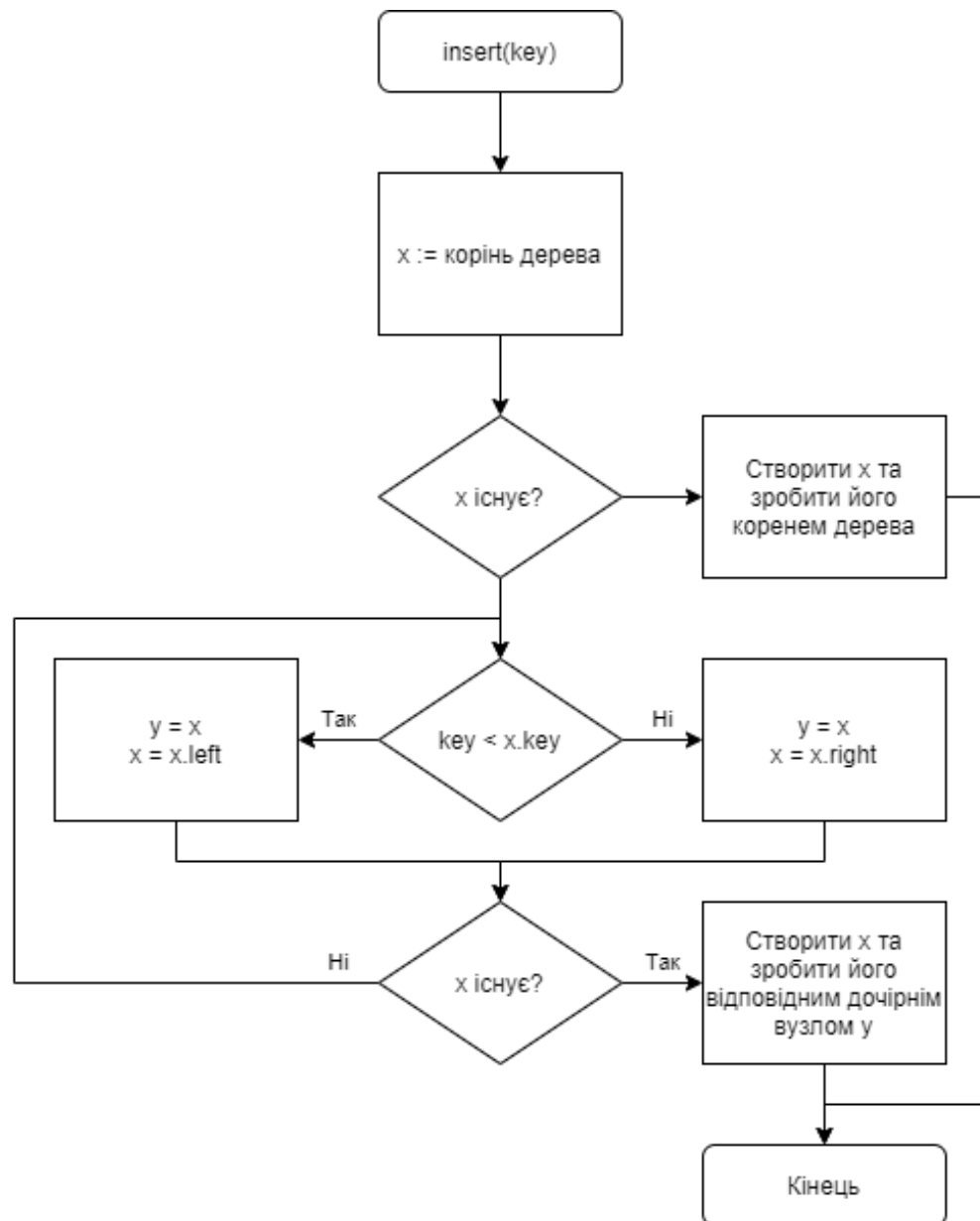


Рис. 2 — Блок-схема алгоритму вставки елемента в бінарне дерево пошуку

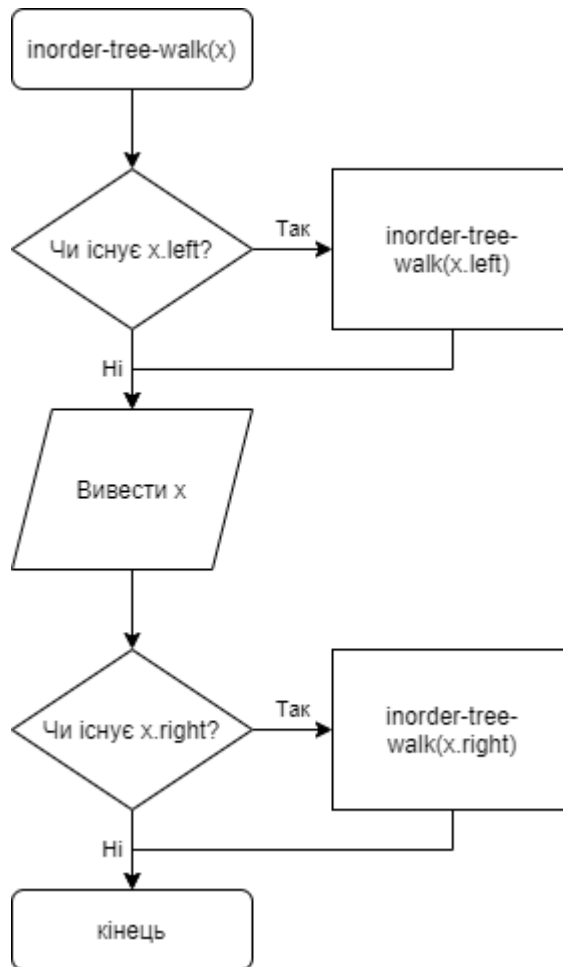


Рис. 3 — Рекурсивне проходження бінарного дерева пошуку

Асимптотична складність такого сортування у середньому випадку складає $O(n \log n)$, оскільки спочатку здійснюється вставка n елементів за $O(\log n)$, а потім робиться проходження дерева по порядку за $O(n)$. Але якщо масив даних уже відсортований або близький до цього, то кожен наступний елемент буде більшим (меншим) за попередній, і дерево по суті буде зв'язним списком, тобто вставка елементів буде відбуватися за $O(n)$ і загальна складність піднімається до $O(n^2)$. Таке бінарне дерево називають виродженим. Для запобігання такого випадку потрібно після кожної вставки елемента проводити балансування дерева. Деревя з такою перевіркою називають збалансованими, і алгоритм сортування бінарним деревом (англ. Tree sort) на таких деревах має асимптотичну складність $O(n \log n)$ у найгіршому випадку.

Загалом, алгоритм сортування бінарним деревом працює схоже на швидке сортування (англ. Quick sort): обирається один елемент, з яким порівнюються всі

інші і відповідно поділяють вихідний масив на дві половини — більша за цей елемент і менша за нього. Але оскільки швидке сортування відбувається in-place, тобто без використання додаткової пам'яті, а також коефіцієнти при $O(n \log n)$ менші, то він у більшості випадків набагато ефективніше за сортування двійковим деревом. Сортування двійковим деревом більш корисне у тих випадках, коли до масиву будуть з часом додаватися нові елементи і потрібно одразу зберігати їх у відсортованому вигляді. Ще одною перевагою над швидким сортуванням є найгірша асимптотична складність $O(n \log n)$ при реалізації на основі збалансованого дерева, але таку саму складність має й описане в пункті 3.2 сортування бінарною купою, при чому він також виконується без використання додаткової пам'яті та з меншими коефіцієнтами.

2.5. Пошук

Одною з найбільш використовуваних структур даних для пошуку є хеш-таблиці. Вони дозволяють проводити пошук за $O(1)$, на відміну від бінарних дерев пошуку. Але якщо є потреба знайти всі елементи з деякого інтервалу, то потрібно буде ітерувати через кожен елемент хеш-таблиці і перевіряти, чи лежить він у потрібному інтервалі, тобто асимптотична складність такого пошуку буде лінійною. Бінарні дерева пошуку дозволяють робити це за логарифмічний час, оскільки ми можемо легко пройти через усі елементи з інтервалу, йдучи наліво, якщо даний елемент більше нижньої границі інтервалу, а коли лівий дочірній елемент уже не буде лежати в інтервалі, то ми не будемо ітерувати через решту елементів дерева наліво від даного. Таким чином ми можемо знайти найменший елемент, що належить інтервалу, за $O(\log n)$, а потім по порядку пройти всі елементи, що належать інтервалу, доки не знайдемо максимальний елемент дерева всередині цього інтервалу. Складність такого алгоритму буде $O(k)$, де k — кількість елементів, що належать інтервалу. Загальна складність такого пошуку буде $O(\log n) + O(k) = O(\log n)$.

Але, як і з сортуванням, потрібно гарантувати, що дерево не буде виродженим. Як уже було сказано в пункті 3.3, для цього використовують збалансовані бінарні дерева [5]. Розглянемо АВЛ-дерево та червоно-чорне дерево.

АВЛ дерево гарантує, що різниця висоти правого і лівого піддерева не більше 1. Це забезпечується тим, що якщо після вставки елемента цей баланс порушено, викликається операція правого чи лівого обертання, яка за $O(1)$ зсуває дерево в один з боків з тим, щоб висоти піддерев зрівнялись [6]. Ілюстрація такого повороту надана на рисунку 4.

У червоно-чорного дерева є 5 основних властивостей:

1. Усі вузли поділяються на червоні та чорні.
2. Корінь дерева чорний.
3. Усі листи чорні.
4. Для кожного вузла кількість чорних вузлів повинна бути однаковою на всіх простих шляхах до листів.
5. Якщо вузол червоний, то обидва дочірніх вузла повинні бути чорними [7].

Підтримання таких властивостей відбувається значно швидше, ніж балансування АВЛ-дерева, оскільки у червоно-чорному дереві в середньому буде відбуватися значно менше поворотів. На відміну від АВЛ-дерева, червоно-чорне дерево гарантує, що висота одного з піддерев не перевищує висоту іншого більше, ніж у 2 рази. Обидва розглянутих види збалансованих дерев ніколи не вироджуються [8].

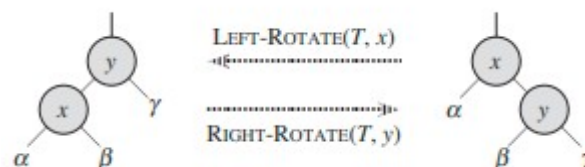


Рис. 4 — Обертання, що використовуються для балансування в АВЛ-дереві та червоно-чорному дереві. Грецькими літерами позначені довільні піддерева

АВЛ дерева балансуються більш жорстко, ніж червоно-чорні, тому пошук елемента за ключем у них відбувається швидше. Але за рахунок більш “вільного”

балансування, червоно-чорні дерева швидше виконують вставку та видалення елементів, тому цей тип збалансованих бінарних дерев став основою для `map`, `multimap` та `multiset` у C++, а також для подібних структур даних у багатьох інших мовах програмування.

2.6. Алгоритм Хафмана

Алгоритм Хафмана — це один з найбільш використовуваних алгоритмів кодування інформації. Він був розроблений Давидом Хафманом, студентом МІТ, і опублікований у 1952 році [9]. Цей алгоритм значно зменшує розмір файлів без втрати інформації за рахунок того, що замість зберігання символів у вигляді однакових 8-бітних кодів незалежно від контексту, він кодує ці символи в унікальні послідовності бітів в залежності від частоти, з якою вони зустрічаються. Тобто якщо символ зустрічається дуже часто, він буде кодуватися малою кількістю бітів, а якщо він зустрічається рідко — то він кодується більше, ніж 8 бітами. При цьому алгоритм Хафмана забезпечує однозначність декодування, тобто одна послідовність бітів завжди може бути декодована тільки одним способом.

Найбільш ефективною реалізацією алгоритму Хафмана є бінарне дерево, відсортоване за частотою, з якою символи зустрічаються в файлі. У даній роботі файл спочатку аналізується та створюється таблиця частот символів у ньому. Далі створюється черга з пріоритетом на основі бінарної купи, у якій вузол з найменшою частотою має найбільший пріоритет. Поки черга не пуста, робляться наступні дії:

1. Видалити два вузла з найбільшим пріоритетом з черги.
2. Створити новий вузол, нащадками якого будуть два витягнуті вузла, та ключ якого є сумою частот цих вузлів.
3. Додати його в чергу.

Складність алгоритму — $O(n \log n)$, оскільки вставка в чергу з пріоритетом займає $O(\log n)$, і таких вставок буде пропорційна n — загальній кількості

унікальних символів у файлі. Після виконання цього алгоритму побудується оптимальне префіксне дерево, яке буде єдиним елементом черги. На рис. 5 наведено неформальний ілюстрований приклад виконання алгоритму.

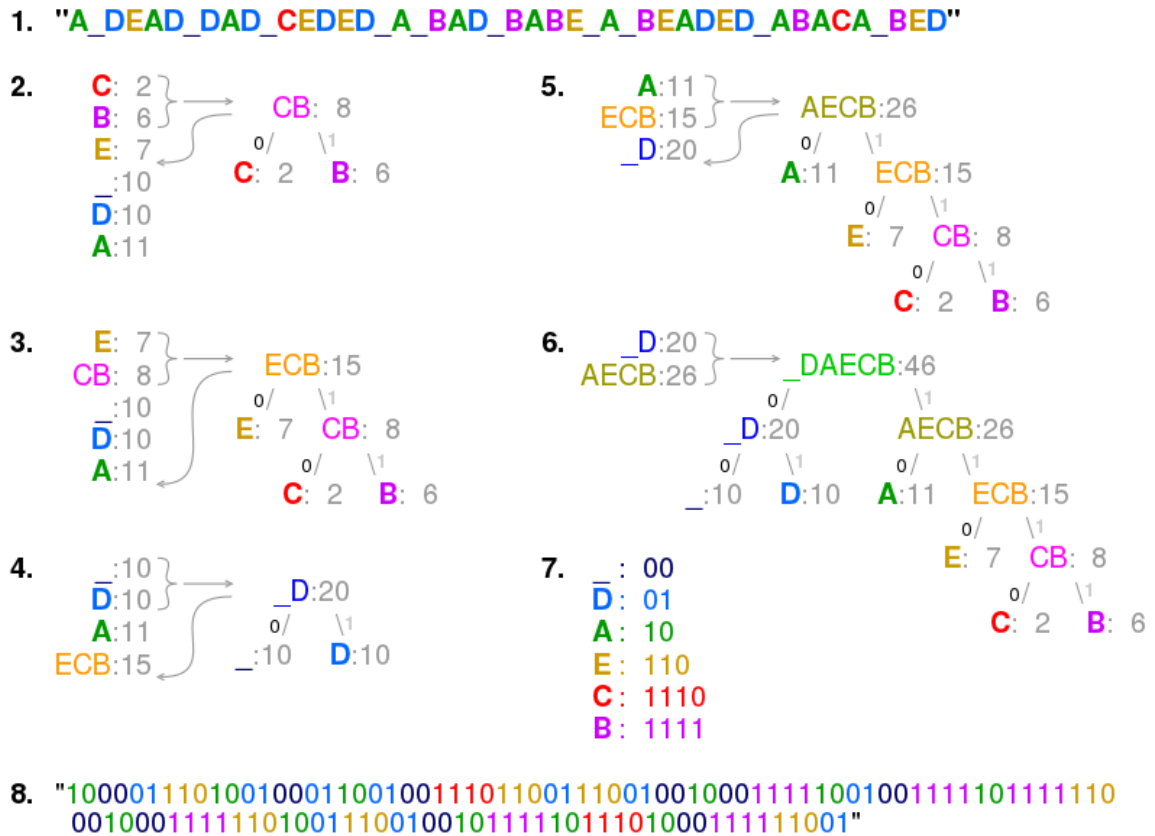


Рис. 5 — Приклад кодування інформації алгоритмом Хафмана

Для того, щоб файл можна було декодувати, в даній роботі було вирішено записати на початку файлу “магічне число” з трьох байтів — “HUF”, яке буде позначати, що цей файл закодовано алгоритмом Хафмана. Далі записується розмір таблиці частот і сама таблиця. Це дозволяє прочитати на початку файла цю таблицю та відтворити префіксне дерево для декодування.

Процес декодування відбувається наступним чином:

1. Створюється вказівник на корінь префіксного дерева.
2. Зчитується один біт з файлу.
3. Якщо прочитана одиниця, то вказівник переміщується до правого нащадка вузла, на який він вказує. Якщо нуль — до лівого.

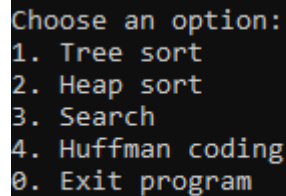
4. Якщо вказівник вказує на лист, то отримати символ, записаний у цьому листі, і записати його у декодований файл, після чого повернути вказівник на корінь дерева.
5. Повторювати п.п. 1 - 4, поки не буде прочитано весь файл.

З точки зору програмування для побітового зчитування файлу довелося створити кілька спеціальних функцій та розглянути деякі специфічні випадки, оскільки файл краще всього читати деякими сегментами в буфер, а потім аналізувати цей буфер. Коли програма доходить до кінця буферу, вона зчитує наступний сегмент і продовжує аналіз.

РОЗДІЛ 3. ОПИС ПРОГРАМНОГО ПРОДУКТУ

3.1. Опис меню програми

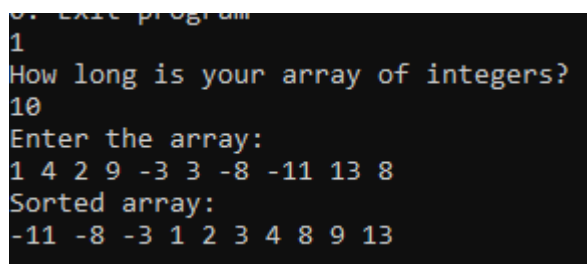
Отже, програма повинна сортувати масиви деревом, купою, шукати елементи, а також кодувати алгоритмом Хафмана. При запуску програми з'являється меню, де є ці 4 варіанти, а також можливість завершити роботу.



```
Choose an option:  
1. Tree sort  
2. Heap sort  
3. Search  
4. Huffman coding  
0. Exit program
```

Рис. 6 — Основне меню програми

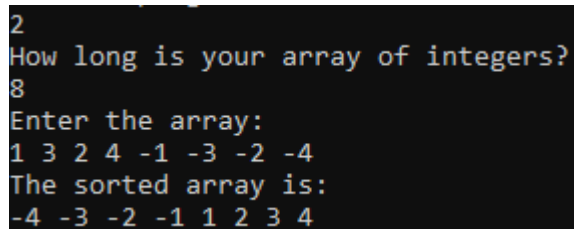
При виборі алгоритму сортування деревом, програма запитує розмір масиву для сортування, а потім сам масив, після чого виводить відсортований масив.



```
0. Exit program  
1  
How long is your array of integers?  
10  
Enter the array:  
1 4 2 9 -3 3 -8 -11 13 8  
Sorted array:  
-11 -8 -3 1 2 3 4 8 9 13
```

Рис. 7 — Приклад виконання алгоритму сортування деревом

Сортування купою оформлено так само:



```
2  
How long is your array of integers?  
8  
Enter the array:  
1 3 2 4 -1 -3 -2 -4  
The sorted array is:  
-4 -3 -2 -1 1 2 3 4
```

Рис. 8 — Приклад виконання алгоритму сортування купою

У випадку з алгоритмом пошуку, користувач спочатку вводить “базу даних”, а потім йому надається можливість вийти у головне меню, перевірити одне число на наявність у базі даних, або вивести всі елементи бази даних, які належать заданому інтервалу.

```

3
Enter the number of elements in your tree: 8
Enter the numbers: 1 2 3 4 5 6 7 8
0. Back to main menu
1. Find a number
2. Find all numbers in range
1
Enter a number: 4
The number is found
0. Back to main menu
1. Find a number
2. Find all numbers in range
1 9
Enter a number: The number is not found
0. Back to main menu
1. Find a number
2. Find all numbers in range
2
Enter the range (for ex. "3 7"): 2 4
2 3 4
0. Back to main menu
1. Find a number
2. Find all numbers in range
0

Choose an option:
1. Tree sort
2. Heap sort
3. Search
4. Huffman coding
0. Exit program

```

Рис. 9 — Приклад виконання алгоритму пошуку

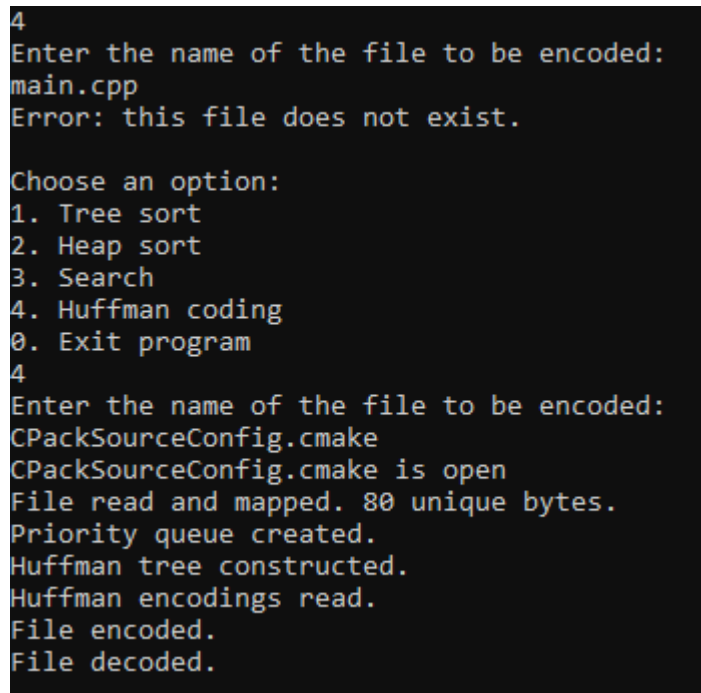
При виборі алгоритму Хафмана, програма спочатку запитує шлях до файлу, потім кодує його у файл `encoded_file` та декодує його у файл `decoded_file`, щоб можна було порівняти його з оригінальним та переконатися в правильності роботи алгоритму. Якщо введений файл не існує, буде виведено відповідне повідомлення.

3.2. Можливі помилки

Слід зауважити, що якщо при введенні масиву чисел у програму ввести деякий розмір масиву X , а потім ввести більше, ніж X чисел, то решта чисел будуть зчитані програмою як команди для наступних меню. Як один з варіантів запобігання таких ситуацій можна запропонувати таке рішення: замість зчитування розміру масиву n і зчитування після цього n чисел у циклі, читати

один рядок повністю (до переходу на наступний рядок), а потім конвертувати його в набір чисел, записуючи їх у динамічний масив. У даній роботі таке рішення не реалізовано, але це можна вважати одною з перспектив покращення програми. Також програма буде поводитись непередбачувано, якщо замість цілих чисел вводити буквенні символи (крім меню). Для запобігання цього у даній програмі після кожної операції введення цілого числа додається наступний код, який виводить помилку в описаній ситуації:

```
if (std::cin.fail())
{
    std::cout << "Error: expected an integer\n";
    exit(EXIT_FAILURE);
}
```



```
4
Enter the name of the file to be encoded:
main.cpp
Error: this file does not exist.

Choose an option:
1. Tree sort
2. Heap sort
3. Search
4. Huffman coding
0. Exit program
4
Enter the name of the file to be encoded:
CPackSourceConfig.cmake
CPackSourceConfig.cmake is open
File read and mapped. 80 unique bytes.
Priority queue created.
Huffman tree constructed.
Huffman encodings read.
File encoded.
File decoded.
```

Рис. 10 — Приклад виконання алгоритму Хафмана. У даному прикладі файл після кодування зменшився на 27%.

3.3. Структура коду програми, спосіб компіляції та запуску

Код програми поділений на 6 файлів .cpp та один файл заголовків .hpp:

- main.cpp – основне меню програми;
- heapsort.cpp – алгоритм сортування купою;
- treesort.cpp – алгоритм сортування бінарним деревом;

- `search.cpp` – червоно-чорне дерево та алгоритм пошуку;
- `priority_queue.cpp` – черга з пріоритетом для алгоритму Хафмана;
- `huffman_coding.cpp` – алгоритм Хафмана;
- `huffman_coding.hpp` – заголовки деяких структур для алгоритму Хафмана.

Також у проекті є файл `CMakeLists.txt`, у якому вказані всі перераховані `.cpp` файли як залежності проекту `kursach1`. Для компіляції проекту потрібно в терміналі зайти за допомогою команди `cd` в папку проекту з файлом `CMakeLists.txt` й запустити генерацію файлів для збирання проекту командою `$ cmake .`

Після цього буде створено `Makefile` проекту і можна запустити компіляцію командою

`$ make`

Для запуску програми потрібно в терміналі в папці з виконуваним файлом дати команду

`$./kursach1`

ВИСНОВКИ

Отже, у даній роботі було досліджено бінарні дерева, їх різновиди та способи використання, а також реалізовано найбільш відомі алгоритми з їх використанням: сортування двійковим деревом, сортування купою, пошук, алгоритм Хафмана. Для алгоритму Хафмана було реалізовано систему запису та зчитування бінарного коду довільної довжини до/з файлу, незалежно від його подільності на 8 (на байти). Було створено програмний продукт з консольним інтерфейсом користувача та меню.

Бінарні дерева дуже широко використовуються в програмуванні, їх реалізації є в стандартних бібліотеках більшості відомих мов програмування. Серед їх застосувань можна назвати наступні:

- пошук в базах даних та інших програмах, які його потребують;
- сортування купою;
- черги з пріоритетом (використовуються в багатьох операційних системах для розкладу процесів, в алгоритмах на пошук шляху, ...);
- алгоритм Хафмана, що використовується в алгоритмах стиснення файлів (у тому числі .jpg і .mp3 форматів);

Також під час виконання роботи я закріпив вивчений за курс матеріал з алгоритмізації та програмування, в тому числі: базова логіка програми (if, else, цикли), створення інтерфейсу, запис та читання файлів, робота з динамічною пам'яттю та вказівниками, поділ програми на функції та модулі, налагодження програми тощо [10].

СПИСОК ЛІТЕРАТУРИ

1. Cormen, Leiserson, Rivest, Stein – Introduction to algorithms, Third edition / Cambridge, Massachusetts: The MIT Press, 2009. С. 151-166, 286-333.
2. Heapsort – Wikipedia, <https://en.wikipedia.org/wiki/Heapsort>
3. Binary search tree – Wikipedia, https://en.wikipedia.org/wiki/Binary_search_tree
4. Tree sort – Wikipedia, https://en.wikipedia.org/wiki/Tree_sort
5. Self-balancing binary search tree - Wikipedia, https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree
6. Data Structures and Algorithms: AVL Trees, <https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html>
7. Data Structures and Algorithms: Red-Black Trees, https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html
8. A time comparison between AVL trees and red black trees / CSREA Press, 2019.
URL: <https://csce.ucmss.com/cr/books/2019/LFS/CSREA2019/FCS2407.pdf>
9. Huffman coding – Wikipedia, https://en.wikipedia.org/wiki/Huffman_coding
10. Bjarne Stroustrup, The C++ programming language, 4th Edition / Addison-Wesley Professional, 2013.

ДОДАТКИ

Додаток А. Текст програми

main.cpp:

```
#include <iostream>

void treesort();
void heapsort();
void search();
void huffman_coding();

int main(int, char **)
{
    char c;

    while (true)
    {
        std::cout << "\nChoose an option:\n"
            << "1. Tree sort\n"
            << "2. Heap sort\n"
            << "3. Search\n"
            << "4. Huffman coding\n"
            << "0. Exit program\n";
        std::cin >> c;
        switch (c)
        {
            case '0':
                return 0;
                break;
            case '1':
                treesort();
                break;
            case '2':
                heapsort();
                break;
            case '3':
                search();
                break;
            case '4':
                huffman_coding();
                break;
            default:
                std::cout << "Invalid option\n";
                break;
        }
    }
}
```

treesort.cpp:

```

#include <iostream>

struct Node
{
    int key;
    Node *left, *right;
    Node(int key);
    void insert(int key);
    void store(int *arr, int &i);
};

Node::Node(int k)
{
    key = k;
    left = right = nullptr;
}

void Node::insert(int k)
{
    if (k < key)
    {
        if (left == nullptr)
        {
            left = new Node(k);
        }
        else
        {
            left->insert(k);
        }
    }
    else
    {
        if (right == nullptr)
        {
            right = new Node(k);
        }
        else
        {
            right->insert(k);
        }
    }
}

void Node::store(int *arr, int &i)
{
    if (left != nullptr)
    {
        left->store(arr, i);
    }
    arr[i] = key;
    i++;
}

```

```

        if (right != nullptr)
        {
            right->store(arr, i);
        }
    }

void treesort()
{
    int len;
    std::cout << "How long is your array of integers?\n";
    std::cin >> len;
    if (std::cin.fail())
    {
        std::cout << "Error: expected an integer\n";
        exit(EXIT_FAILURE);
    }
    int *array = new int[len];
    std::cout << "Enter the array:\n";
    for (int i = 0; i < len; i++)
    {
        std::cin >> array[i];
        if (std::cin.fail())
        {
            std::cout << "Error: expected an integer\n";
            exit(EXIT_FAILURE);
        }
    }

    Node *root = new Node(array[0]);
    for (int i = 1; i < len; i++)
    {
        root->insert(array[i]);
    }

    int i = 0;
    root->store(array, i);

    std::cout << "Sorted array:\n";
    for (int i = 0; i < len; i++)
    {
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;
}

```

heapsort.cpp:

```

#include <iostream>

int parent(int i)
{
    return (i - 1) / 2;
}

```

```

}

int left(int i)
{
    return i * 2 + 1;
}

int right(int i)
{
    return i * 2 + 2;
}

void siftDown(int *arr, int n, int i)
{
    int l = left(i);
    int r = right(i);
    int smallest;
    if ((l < n) && (arr[l] < arr[i]))
        smallest = l;
    else
        smallest = i;
    if ((r < n) && (arr[r] < arr[smallest]))
        smallest = r;
    if (smallest != i)
    {
        std::swap(arr[i], arr[smallest]);
        siftDown(arr, n, smallest);
    }
}

void heapsort()
{
    int len;

    std::cout << "How long is your array of integers?\n";
    std::cin >> len;
    if (std::cin.fail())
    {
        std::cout << "Error: expected an integer\n";
        exit(EXIT_FAILURE);
    }
    int *array = new int[len];
    std::cout << "Enter the array:\n";
    for (int i = 0; i < len; i++)
    {
        std::cin >> array[i];
        if (std::cin.fail())
        {
            std::cout << "Error: expected an integer\n";
            exit(EXIT_FAILURE);
        }
    }
}

```

```

    }
    for (int i = (len - 1) / 2; i >= 0; i--)
    {
        siftDown(array, len, i);
    }
    for (int i = len - 1; i > 0; i--)
    {
        std::swap(array[i], array[0]);
        siftDown(array, i, 0);
    }
    std::cout << "The sorted array is:\n";
    for (int i = len - 1; i >= 0; i--)
    {
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;
}

```

search.cpp:

```

#include <iostream>
#include <vector>

using namespace std;

enum Color
{
    BLACK, RED
};

struct Node
{
    Node *left, *p, *right;
    int data;
    enum Color color;
    Node();
};

Node* nil = new Node();
Node* root = new Node();

Node::Node()
{
    left = nil;
    right = nil;
    p = nil;
    color = BLACK;
    data = 0;
}

Node* find(int obj)
{

```



```

Node* x = root;
while ((x != nil) && (x->data != obj))
{
    if (obj < x->data)
    {
        x = x->left;
    }
    else
    {
        x = x->right;
    }
}
if (x == nil)
{
    x = nullptr;
}
return x;
}

void inOrderTreeWalk(Node* x, int num1, int num2, vector<int>& vec)
{
    if (x == nil)
    {
        return;
    }

    if (x->left != nil && x->data >= num1)
    {
        inOrderTreeWalk(x->left, num1, num2, vec);
    }

    if (x->data >= num1 && x->data <= num2)
    {
        vec.push_back(x->data);
    }

    if (x->right != nil && x->data <= num2)
    {
        inOrderTreeWalk(x->right, num1, num2, vec);
    }
}

vector<int> findInRange(int num1, int num2)
{
    vector<int> vec;
    inOrderTreeWalk(root, num1, num2, vec);
    return vec;
}

void leftRotate(Node* x)
{

```

```

    if ((x == nil) || (x->right == nil))
    {
        return;
    }
    Node* node = x->right;
    x->right = node->left;
    if (node->left != nil)
    {
        node->left->p = x;
    }
    node->p = x->p;
    if (x->p == nil)
    {
        root = node;
    }
    else if (x == x->p->left)
    {
        x->p->left = node;
    }
    else
    {
        x->p->right = node;
    }
    node->left = x;
    x->p = node;
}

```

```

void rightRotate(Node* x)
{
    if ((x == nil) || (x->left == nil))
    {
        return;
    }
    Node* node = x->left;
    x->left = node->right;
    if (node->right != nil)
    {
        node->right->p = x;
    }
    node->p = x->p;
    if (x->p == nil)
    {
        root = node;
    }
    else if (x == x->p->right)
    {
        x->p->right = node;
    }
    else
    {
        x->p->left = node;
    }
}

```

```

    }
    node->right = x;
    x->p = node;
}

void insertFixup(Node* x)
{
    Node* uncle;
    while (x->p->color == RED)
    {
        if (x->p == x->p->p->left)
        {
            uncle = x->p->p->right;
            if (uncle->color == RED)
            {
                x->p->color = BLACK;
                uncle->color = BLACK;
                x->p->p->color = RED;
                x = x->p->p;
            }
            else
            {
                if (x == x->p->right)
                {
                    x = x->p;
                    leftRotate(x);
                }
                x->p->color = BLACK;
                x->p->p->color = RED;
                rightRotate(x->p->p);
            }
        }
        else if (x->p == x->p->p->right)
        {
            uncle = x->p->p->left;
            if (uncle->color == RED)
            {
                x->p->color = BLACK;
                uncle->color = BLACK;
                x->p->p->color = RED;
                x = x->p->p;
            }
            else
            {
                if (x == x->p->left)
                {
                    x = x->p;
                    rightRotate(x);
                }
                x->p->color = BLACK;
                x->p->p->color = RED;
            }
        }
    }
}

```

```

        leftRotate(x->p->p);
    }
}
root->color = BLACK;
}

void insert(int obj)
{
    if (find(obj))
    {
        return;
    }

    Node* node, * prevNode;
    Node* newNode = new Node;
    newNode->left = nil;
    newNode->right = nil;
    newNode->p = nil;
    newNode->data = obj;
    newNode->color = RED;
    prevNode = nil;
    node = root;
    while (node != nil)
    {
        prevNode = node;
        if (newNode->data < node->data)
        {
            node = node->left;
        }
        else
        {
            node = node->right;
        }
    }
    newNode->p = prevNode;
    if (prevNode == nil)
    {
        root = newNode;
    }
    else if (newNode->data < prevNode->data)
    {
        prevNode->left = newNode;
    }
    else
    {
        prevNode->right = newNode;
    }
    newNode->color = RED;
    insertFixup(newNode);
}

```

```

void search()
{
    vector<int> vec;
    std::cout << "Enter the number of elements in your tree: ";
    int size;
    std::cin >> size;
    if (std::cin.fail())
    {
        std::cout << "Error: expected an integer\n";
        return;
    }
    std::cout << "Enter the numbers: ";
    int num, num1, num2;
    for (int i = 0; i < size; i++)
    {
        std::cin >> num;
        if (std::cin.fail())
        {
            std::cout << "Error: expected an integer\n";
            return;
        }
        insert(num);
    }

    char c;
    while (true)
    {
        std::cout << "0. Back to main menu\n";
        std::cout << "1. Find a number\n";
        std::cout << "2. Find all numbers in range\n";
        std::cin >> c;
        switch (c)
        {
            case '0':
                return;
                break;
            case '1':
                std::cout << "Enter a number: ";
                std::cin >> num;
                if (std::cin.fail())
                {
                    std::cout << "Error: expected an integer\n";
                    exit(EXIT_FAILURE);
                }
                if (find(num))
                {
                    std::cout << "The number is found\n";
                }
                else
                {

```

```

        std::cout << "The number is not found\n";
    }
    break;
case '2':
    std::cout << "Enter the range (for ex. \"3 7\") : ";
    std::cin >> num1 >> num2;
    if (std::cin.fail())
    {
        std::cout << "Error: expected an integer\n";
        exit(EXIT_FAILURE);
    }
    vec = findInRange(num1, num2);
    for (int i : vec)
    {
        std::cout << i << " ";
    }
    std::cout << std::endl;
    break;
default:
    std::cout << "Invalid choice\n";
    break;
}
}
}

```

huffman_coding.hpp:

```

#ifndef _HUFFMAN_CODING_HPP
#define _HUFFMAN_CODING_HPP

#include <unordered_map>
#include <bitset>
#include <vector>

struct bits
{
    short bits;
    char length;
};

struct TreeNode
{
    TreeNode(std::pair<char, int> p);
    std::pair<char, int> data;
    TreeNode *right;
    TreeNode *left;
};

class PriorityQueue
{
    std::vector<TreeNode *> vec;

```

```

public:
    int parent(int index);
    int left(int index);
    int right(int index);
    void minHeapify(int index);
    void insert(TreeNode *element);
    TreeNode *minimum();
    TreeNode *extractMin();

    TreeNode *constructHuffmanTree();
    std::unordered_map<char, bits> *getHuffmanEncodings();

private:
    void recursiveEncode(TreeNode *node, bits *b,
std::unordered_map<char, bits> *map);
};

#endif

```

priority_queue.cpp:

```

#include <algorithm>
#include "huffman_coding.hpp"

TreeNode::TreeNode(std::pair<char, int> p)
{
    right = nullptr;
    left = nullptr;
    data = p;
}

int PriorityQueue::parent(int i)
{
    return (i - 1) / 2;
}

int PriorityQueue::left(int i)
{
    return 2 * i + 1;
}

int PriorityQueue::right(int i)
{
    return 2 * i + 2;
}

void PriorityQueue::minHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest;
    if ((l < vec.size()) && (vec[l]->data.second < vec[i]-

```

```

>data.second))
    smallest = 1;
else
    smallest = i;
    if ((r < vec.size()) && (vec[r]->data.second < vec[smallest]-
>data.second))
        smallest = r;
    if (smallest != i)
    {
        std::swap(vec[i], vec[smallest]);
        minHeapify(smallest);
    }
}

void PriorityQueue::insert(TreeNode *element)
{
    vec.push_back(element);
    int i = vec.size() - 1;
    while ((i > 0) && (vec[parent(i)]->data.second > vec[i]-
>data.second))
    {
        std::swap(vec[i], vec[parent(i)]);
        i = parent(i);
    }
}

TreeNode *PriorityQueue::minimum()
{
    return vec[0];
}

TreeNode *PriorityQueue::extractMin()
{
    if (vec.size() < 1)
    {
        return nullptr;
    }

    TreeNode *min = vec[0];
    vec[0] = vec[vec.size() - 1];
    vec.resize(vec.size() - 1);
    minHeapify(0);
    return min;
}

TreeNode *PriorityQueue::constructHuffmanTree()
{
    TreeNode *node;
    TreeNode *first;
    TreeNode *second;
    while (vec.size() > 1)

```



```

    {
        first = extractMin();
        second = extractMin();
        node = new TreeNode(std::pair<char, int>(0, first->data.second + second->data.second));
        node->left = first;
        node->right = second;
        insert(node);
    }
    return minimum();
}

std::unordered_map<char, bits> *PriorityQueue::getHuffmanEncodings()
{
    std::unordered_map<char, bits> *map = new
std::unordered_map<char, bits>();

    TreeNode *currentNode = vec[0];
    bits *b = new bits;
    b->bits = 0;
    b->length = 0;

    recursiveEncode(vec[0], b, map);

    return map;
}

void PriorityQueue::recursiveEncode(TreeNode *node, bits *b,
std::unordered_map<char, bits> *map)
{
    if ((node->left == nullptr) && (node->right == nullptr))
    {
        map->insert(std::pair<char, bits>(node->data.first, *b));
        return;
    }
    if (node->left != nullptr)
    {
        b->length++;
        recursiveEncode(node->left, b, map);
        b->length--;
    }
    if (node->right != nullptr)
    {
        b->bits |= (1 << (15 - b->length));
        b->length++;
        recursiveEncode(node->right, b, map);
        b->length--;
        b->bits &= ~(1 << (15 - b->length));
    }
    return;
}

```

huffman_coding.cpp:

```
#include <iostream>
#include <fstream>
#include <filesystem>
#include "huffman_coding.hpp"

using namespace std;

const string kEncodedFileName = "encoded_file";
const string kDecodedFileName = "decoded_file";
const char kMagicCode[3] = {'h', 'u', 'f'};

void writeInt(int &number, ofstream &os)
{
    os.write(reinterpret_cast<char *>(&number), 4);
}

int readInt(ifstream &is)
{
    int number;
    is.read(reinterpret_cast<char *>(&number), 4);
    return number;
}

void writeShort(short &number, ofstream &os)
{
    os.write(reinterpret_cast<char *>(&number), 2);
}

void encode(ifstream &is, ofstream &os, unordered_map<char, bits>
*encodings, int file_length)
{
    char outbyte;
    char length = 0;
    char ptr;
    bits current;

    for (auto c : kMagicCode)
        os << c;

    writeInt(file_length, os);
    int tablesize = encodings->size();
    writeInt(tablesize, os);
    for (auto el : *encodings)
    {
        os << el.first;
        writeShort(el.second.bits, os);
        os << el.second.length;
    }

    static int st = 0;
```

```

    outbyte = 0;
    char *buf = new char[file_length];
    char *obuf = new char[file_length];
    int optr = 0;
    is.read(buf, file_length);
    for (int i = 0; i < file_length; i++)
    {
        ptr = 0;
        current = encodings->at(buf[i]);
        st++;
        while (ptr < current.length)
        {
            outbyte |= (int)((bool) (current.bits & (1 << (15 -
ptr)))) << (7 - length));
            ptr++;
            length++;
            if (length == 8)
            {
                obuf[optr] = outbyte;
                optr++;
                outbyte = 0;
                length = 0;
            }
        }
        obuf[optr] = outbyte;
        optr++;
        os.write(obuf, optr);
        delete[] buf;
        delete[] obuf;
    }
}

```

```

int decode(ifstream &is, ofstream &os, int file_length)
{
    int length;
    int tablesize;
    char inbyte;
    TreeNode *tree = new TreeNode(pair<char, int>(0, 0));
    TreeNode *currentTreeNode = tree;
    short byte;
    char bytelen;

    for (auto c : kMagicCode)
    {
        is >> inbyte;
        if (inbyte != c)
            return -1;
    }

    length = readInt(is);
    tablesize = readInt(is);
}

```

```

char *table = new char[tablesize * 4];
is.read(table, tablesize * 4);

char c;
int st = 0;
for (int i = 0; i < tablesize; i++)
{
    c = table[i * 4 + 0];
    byte = *reinterpret_cast<short *>(&table[i * 4 + 1]);
    bytelen = table[i * 4 + 3];

    for (int j = 0; j < bytelen; j++)
    {
        if (byte & (1 << (15 - j)))
        {
            if (currentTreeNode->right == nullptr)
            {currentTreeNode->right = new
TreeNode(pair<char, int>(0, 0)); st++;}
            currentTreeNode = currentTreeNode->right;
        }
        else
        {
            if (currentTreeNode->left == nullptr)
            {currentTreeNode->left = new TreeNode(pair<char,
int>(0, 0)); st++;}
            currentTreeNode = currentTreeNode->left;
        }
    }
    currentTreeNode->data.first = c;
    currentTreeNode = tree;
}
delete[] table;

int infolength = file_length - sizeof(kMagicCode) - 2 * 4 -
tablesize * 4;
char *buf = new char[infolength];
char *obuf = new char[length];
int original_length = length;
int ptr = 0;
is.read(buf, infolength);
for (int i = 0; i < infolength; i++)
{
    for (int j = 7; (j >= 0) && (length > 0); j--)
    {
        if (buf[i] & (1 << j))
        {
            if (currentTreeNode->right == nullptr)
            {
                obuf[ptr] = currentTreeNode->data.first;
                ptr++;
                length--;
            }
        }
    }
}

```

```

        currentTreeNode = tree->right;
    }
    else
    {
        currentTreeNode = currentTreeNode->right;
    }
}
else
{
    if (currentTreeNode->left == nullptr)
    {
        obuf[ptr] = currentTreeNode->data.first;
        ptr++;
        length--;
        currentTreeNode = tree->left;
    }
    else
    {
        currentTreeNode = currentTreeNode->left;
    }
}
}
}
if (length != 0) return -1;
os.write(obuf, original_length);
delete[] buf;
delete[] obuf;
return 0;
}

```

```

void huffman_coding()
{
    unordered_map<char, int> map;

    string filename;
    cout << "Enter the name of the file to be encoded:\n";
    cin.get();
    getline(cin, filename);

    // open the original file
    ifstream originalFile;
    originalFile.open(filename);
    if (!originalFile.is_open())
    {
        cout << "Error: this file does not exist.\n";
        return;
    }
    int file_length = filesystem::file_size(filename);
    cout << filename << " is open\n";

    // read the file byte by byte and count frequencies of each byte

```

```

char *buf;
buf = new char[file_length];
originalFile.read(buf, file_length);
for (int i = 0; i < file_length; i++)
{
    if (!map.count(buf[i]))
    {
        map.insert(pair<char, int>(buf[i], 1));
    }
    else
    {
        map.at(buf[i])++;
    }
}
delete[] buf;
cout << "File read and mapped. " << map.size() << " unique
bytes.\n";

// transfer all the resulting bytes and frequencies into a
priority queue
PriorityQueue queue;
TreeNode *node;
for (auto el : map)
{
    node = new TreeNode(el);
    queue.insert(node);
}
cout << "Priority queue created.\n";

// construct Huffman tree and get the encodings in a hash table
queue.constructHuffmanTree();
cout << "Huffman tree constructed.\n";
unordered_map<char, bits> *encodings =
queue.getHuffmanEncodings();
cout << "Huffman encodings read.\n";

// go to the beginning of the file to read it again (for
encoding)
originalFile.clear();
originalFile.seekg(0, originalFile.beg);

// open for writing the future encoded file
ofstream encodedFileO;
encodedFileO.open(kEncodedFileName);
if (!encodedFileO.is_open())
{
    cout << "Error opening encoded file stream.\n";
    return;
}

// encode the original file into the encoded file

```

```

encode(originalFile, encodedFileO, encodings, file_length);
cout << "File encoded.\n";
originalFile.close();
encodedFileO.close();

// open encoded file for reading and decoded file for writing
ifstream encodedFile;
encodedFile.open(kEncodedFileName);
if (!encodedFile.is_open())
{
    cout << "Error opening encoded file.\n";
    return;
}
int encodedFileSize = filesystem::file_size(kEncodedFileName);

ofstream decodedFile;
decodedFile.open(kDecodedFileName);
if (!decodedFile.is_open())
{
    cout << "Error opening decoded file.\n";
    return;
}

// decode the file back
if (decode(encodedFile, decodedFile, encodedFileSize) != -1)
    cout << "File decoded.\n";
else
    cout << "Invalid encoded file.\n";
encodedFile.close();
decodedFile.close();
}

```